



2017 级

《物联网数据存储与管理》课程

# 实 验 报 告

姓 名 王家树

学 号 U201714700

班 号 物联网 1701 班

日 期 2020.05.29

# 目 录

一、实验目的.....	1
二、实验背景.....	1
三、实验环境.....	1
四、实验内容.....	1
4.1 对象存储技术实践.....	2
4.2 对象存储性能分析.....	2
五、实验过程.....	2
5.1 实验环境配置.....	2
5.2 对象存储技术实践.....	3
5.2.1 minio 和 minio client 服务搭建.....	3
5.2.2 Openstack Swift 搭建.....	5
5.3 对象存储性能分析.....	8
六、实验总结.....	11
参考文献.....	11

## 一、实验目的

1. 熟悉对象存储技术，代表性系统及其特性；
2. 实践对象存储系统，部署实验环境，进行初步测试；
3. 基于对象存储系统，架设实际应用，示范主要功能。

## 二、实验背景

对象存储，也叫做基于对象的存储，是用来描述解决和处理离散单元的方法的通用术语，这些离散单元被称作为对象。

就像文件一样，对象包含数据，但是和文件不同的是，对象在一个层结构中不会再有层级结构。每个对象都在一个被称作存储池的扁平地址空间的同一级别里，一个对象不会属于另一个对象的下一级。

对象存储系统（Object-Based Storage System）是综合了 NAS 和 SAN 的优点，同时具有 SAN 的高速直接访问和 NAS 的数据共享等优势，提供了高可靠性、跨平台性以及安全的数据共享的存储体系结构。

Minio 是 Apache License v2.0 下发布的对象存储服务器。它与 Amazon S3 云存储服务兼容。它最适合存储非结构化数据，如照片，视频，日志文件，备份和容器/VM 映像。对象的大小可以从几 KB 到最大 5TB

Swift 最初是由 Rackspace 公司开发的高可用分布式对象存储服务，并于 2010 年贡献给 OpenStack 开源社区作为其最初的核心子项目之一，为其 Nova 子项目提供虚拟机镜像存储服务。Swift 构筑在比较便宜的标准硬件存储基础设施之上，无需采用 RAID（磁盘冗余阵列），通过在软件层面引入一致性散列技术和数据冗余性，牺牲一定程度的数据一致性来达到高可用性和可伸缩性，支持多租户模式、容器和对象读写操作，适合解决互联网的应用场景下非结构化数据存储问题。

## 三、实验环境

实验平台	环境与版本	
本地环境操作系统	Ubuntu 20.04 LTS Linux version 5.4.0-26-generic (bulld@lcy01-amd64-029)	
本地硬件配置	Intel® Core™ i5-7300HQ CPU @ 2.50GHz / 4.9GB	
对象存储服务端	Openstack Swift (docker version: 19.03.9)	Minio
对象存储客户端	python-swiftclient 3.9.0	Minio client
对象存储评测工具	S3 bench	
Python 版本	Python 3.7.6	
Java 版本	openjdk version "1.8.0_252"	
Go 版本	go version go1.14.3 linux/amd64	

## 四、实验内容

本实验的主要内容有，熟悉实验环境并在本地配置环境，通过在本本地配置对象存储服务端与客户端，来熟悉和尝试使用对象存储的技术，同时使用标准的评测工具来对实验环境进行评测，分析对象存储的性能，探索优化速率的方法。

## 4.1 对象存储技术实践

1. 熟悉 git 和 github 操作以及配置 java, python, go 语言相关环境。
2. 下载并安装 minio 服务器端, 完成配置, 开启服务器; 下载安装 minio client 并运用 mc 来对服务端进行功能上的测试。
3. 在本地环境上安装 Docker, 并在 Docker 上配置并部署一个单节点的 Openstack Swift 服务端。在本地环境上安装 python-swiftclient 客户端, 并通过该客户端来对服务端进行功能上的测试。

## 4.2 对象存储性能分析

编写测试脚本, 利用 s3 bench 对实验环境进行测试, 观察运行结果并分析对象存储性能的情况。

### 1. 测试对象尺寸对运行结果的影响。

将负载样例中的 NumClient 和 NumSample 数目保持不变, 对象尺寸逐渐增大, 观察运行结果。

### 2. 测试连接数量、对象数量对运行结果的影响

## 五、实验过程

### 5.1 实验环境配置

老师给的实验介绍和指导篇里面已经给了几种办法了, 这里就简单说一下: python 是安装 anaconda3 来安装 python3 的环境; java 是直接用 Ubuntu 的包管理工具 apt, 用命令 `sudo apt-get install openjdk-8-jdk` 来安装配置 openjdk8 的环境; go 稍微麻烦点, 首先在 go 语言中文资料网下载源代码的压缩包, 然后解压至系统文件 bin 目录下, 设置好环境变量即可。最终效果如下:

```
(base) wang@wanglinux:~/桌面$ python --version
Python 3.7.6
(base) wang@wanglinux:~/桌面$ java -version
openjdk version "1.8.0_252"
OpenJDK Runtime Environment (build 1.8.0_252-8u252-b09-1ubuntu1-b09)
OpenJDK 64-Bit Server VM (build 25.252-b09, mixed mode)
(base) wang@wanglinux:~/桌面$ go version
go version go1.14.3 linux/amd64
(base) wang@wanglinux:~/桌面$ go env
GO111MODULE="on"
GOARCH="amd64"
GOBIN=""
GOCACHE="/home/wang/.cache/go-build"
GOENV="/home/wang/.config/go/env"
```

图 1 实验环境图

而 swift 是基于 docker 安装配置的, docker 安装网络上有很详细的教程, 这里不再赘述, 最终环境如下:

```
(base) wang@wanglinux:~/桌面$ docker version
Client: Docker Engine - Community
 Version:      19.03.9
 API version:  1.40
 Go version:   go1.13.10
 Git commit:   9d988398e7
 Built:        Fri May 15 00:25:20 2020
 OS/Arch:     linux/amd64
 Experimental: false

Server: Docker Engine - Community
 Engine:
  Version:      19.03.9
  API version:  1.40 (minimum version 1.12)
  Go version:   go1.13.10
  Git commit:   9d988398e7
  Built:        Fri May 15 00:23:53 2020
  OS/Arch:     linux/amd64
  Experimental: false
 containerd:
  Version:      1.2.13
  GitCommit:    7ad184331fa3e55e52b890ea95e65ba581ae3429
 runc:
  Version:      1.0.0-rc10
  GitCommit:    dc9208a3303feef5b3839f4323d9beb36df0a9dd
 docker-init:
  Version:      0.18.0
  GitCommit:    fec3683
```

图 2 docker 信息

## 5.2 对象存储技术实践

### 5.2.1 minio 和 minio client 服务搭建

首先配置 Minio 服务端，根据官网上的教程下载安装 minio server:

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
```

使用老师提供的启动脚本一键配置启动:

```
(base) wang@wanglinux:~/test$ sh run-minio.sh

You are running an older version of MinIO released 1 week ago
Update: Run `mc admin update`

Endpoint: http://10.0.2.15:9000 http://172.17.0.1:9000 http://127.0.0.1:9000
AccessKey: hust
SecretKey: hust_obs

Browser Access:
http://10.0.2.15:9000 http://172.17.0.1:9000 http://127.0.0.1:9000

Command-line Access: https://docs.min.io/docs/minio-client-quickstart-guide
$ mc config host add myminio http://10.0.2.15:9000 hust hust_obs

Object API (Amazon S3 compatible):
Go: https://docs.min.io/docs/golang-client-quickstart-guide
Java: https://docs.min.io/docs/java-client-quickstart-guide
Python: https://docs.min.io/docs/python-client-quickstart-guide
JavaScript: https://docs.min.io/docs/javascript-client-quickstart-guide
.NET: https://docs.min.io/docs/dotnet-client-quickstart-guide
```

图 3 minio 服务端启动信息

可以看到他已经输出了访问地址和脚本设定好了的 accesskey 和 secretkey，为了验证服务器是否搭建成功，由于 Minio 提供了 HTML 实现的图形界面，所以我们可以打开浏览器来验证，

直接访问 10.0.2.15:9000。输入上面配置好的密 钥，就可以进入 Minio Browser。:

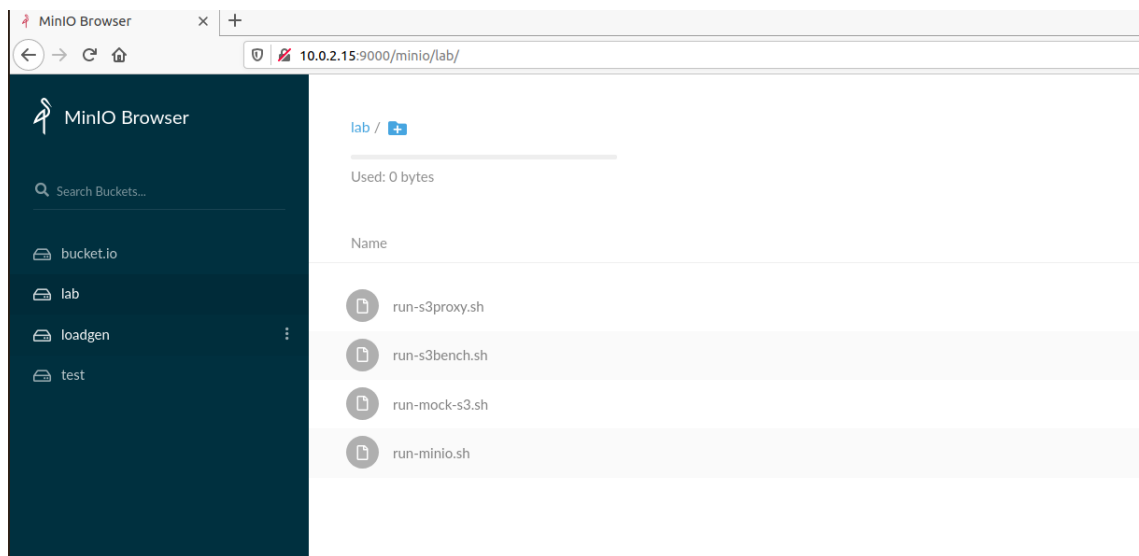


图 4 minio web 界面

此时可以选择右下角红色标记随意添加 bucket 和上传文件。

然后配置 mc client，输入如下命令：

```
wget https://dl.minio.io/client/mc/release/linux-amd64/mc
chmod +x mc
./mc -help
```

可以看到：

```
(base) wang@wanglinux:~/test$ ./mc --help
NAME:
  mc - MinIO Client for cloud storage and filesystems.

USAGE:
  mc [FLAGS] COMMAND [COMMAND FLAGS | -h] [ARGUMENTS...]

COMMANDS:
  ls          list buckets and objects
  mb          make a bucket
  rb          remove a bucket
  cp          copy objects
  mirror      synchronize object(s) to a remote site
  cat         display object contents
  head        display first 'n' lines of an object
  pipe        stream STDIN to an object
  share       generate URL for temporary access to an object
  find        search for objects
  sql         run sql queries on objects
  stat        show object metadata
  mv          move objects
  tree        list buckets and objects in a tree format
  du          summarize disk usage recursively
  lock        set and get object lock configuration
  retention   set retention for object(s)
  legalhold   set legal hold for object(s)
  diff        list differences in object name, size, and date between two buckets
  rm          remove objects
  event       configure object notifications
  ilm         configure bucket lifecycle
  watch       listen for object notification events
  policy      manage anonymous access to buckets and objects
  tag         manage tags for bucket(s) and object(s)
  admin       manage MinIO servers
  config      configure MinIO client
  update      update mc to latest release

GLOBAL FLAGS:
  --autocomplete          install auto-completion for your shell
  --config-dir value, -C value path to configuration folder (default: "/home/wang/.mc")
  --quiet, -q            disable progress bar display
  --no-color              disable color theme
  --json                 enable JSON formatted output
  --debug                enable debug output
  --insecure              disable SSL certificate verification
  --help, -h             show help
  --version, -v          print the version

TIP:
  Use 'mc --autocomplete' to enable shell auto completion
```

图 5 mc 帮助信息

使用命令

```
./mc config host add myminio http://10.0.2.15:9000 hust hust_obs
```

```
./mc mb myminio/lab
./mc cp ./run-minio.sh myminio/lab
./mc ls myminio/lab
```

来连接服务端，并上传文件，查看对象服务器桶内数据，结果如下：

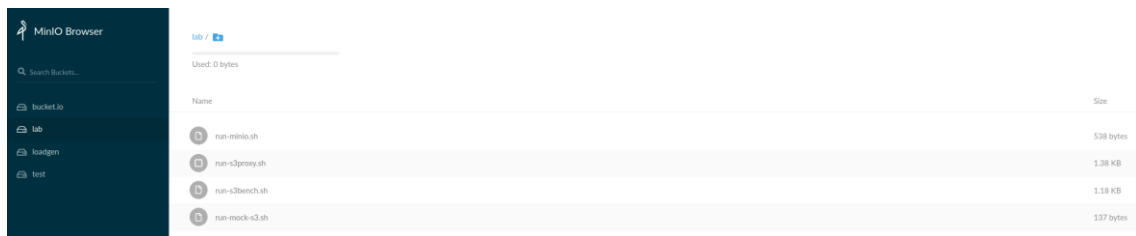
```
(base) wang@wanglinux:~/test$ ./mc config host add myminio http://10.0.2.15:9000 hust hust_obs
Added 'myminio' successfully.
```

```
(base) wang@wanglinux:~/test$ ./mc mb myminio/lab
Bucket created successfully 'myminio/lab'.

(base) wang@wanglinux:~/test$ ./mc ls myminio/lab
[2020-05-27 17:34:23 CST]    538B run-minio.sh
[2020-05-27 17:35:05 CST]    137B run-mock-s3.sh
[2020-05-27 17:35:05 CST]  1.2KiB run-s3bench.sh
[2020-05-27 17:35:05 CST]  1.4KiB run-s3proxy.sh
```

图 6, 7, 8 minio client 功能检测

而通过服务端的浏览器界面可以检查上述操作成功。



Name	Size
run-minio.sh	538 bytes
run-s3proxy.sh	1.38 KB
run-s3bench.sh	1.18 KB
run-mock-s3.sh	137 bytes

图 9 结果检验

## 5.2.2 Openstack Swift 搭建

首先克隆实验文档提供的 Github 仓库，执行如下命令：

```
git clone https://github.com/cs-course/openstack-swift-docker.git
```

接着启动 Docker 服务：

```
systemctl start docker
```

执行后可以检查 docker.service 是否正常启动，启动后可以开始构建 Openstack Swift 的容器。

执行以下命令构建 docker 镜像：

```
Sudo docker build -t openstack-swift-docker .
```

系统根据 Dockerfile 中的步骤执行自动的环境配置与构建：

```
(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo docker build -t openstack-swift-docker .
Sending build context to Docker daemon 126kB
Step 1/17 : FROM phusion/baseimage:0.9.22
0.9.22: Pulling from phusion/baseimage
Image docker.io/phusion/baseimage:0.9.22 uses outdated schema1 manifest format. Please upgrade to a schema2 image for better future compatibility. More
22ecafbbcc4a: Pull complete
80435e0a080: Pull complete
8321ffdf10031: Pull complete
08b8f28a13c2: Pull complete
b401702069a: Pull complete
3ed95cae02: Pull complete
ae027dcdc0e: Pull complete
93bc98227159: Pull complete
Digest: sha256:3c354279a393ceeff1b39f2028892f2dae6843e0eba7706ca29765e47277606
Status: Downloaded newer image for phusion/baseimage:0.9.22
--> 877509368a8d
Step 2/17 : MAINTAINER Zhan.Shi <g.shizhan.g@gmail.com>
--> Running in d9cefed625b0
Removing intermediate container d9cefed625b0
--> 629eb1a7a1e5
Step 3/17 : RUN apt-get update && apt-get install -y mencached rsync pwgen supervisor python-xattr python-mencache python-netifaces
keystoneclient
--> Running in 337db36d1710
get:1 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
get:2 http://security.ubuntu.com/ubuntu xenial-security InRelease [109 kB]
get:3 http://security.ubuntu.com/ubuntu xenial-security/main Sources [206 kB]
get:4 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [109 kB]
get:5 http://security.ubuntu.com/ubuntu xenial-security/restricted Sources [2,243 B]
get:6 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [146 kB]
get:7 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [107 kB]
get:8 http://archive.ubuntu.com/ubuntu xenial/main Sources [1,103 kB]
get:9 http://security.ubuntu.com/ubuntu xenial-security/multiverse Sources [3,517 B]
get:10 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [1,116 kB]
get:11 http://security.ubuntu.com/ubuntu xenial-security/restricted amd64 Packages [12.7 kB]
get:12 http://security.ubuntu.com/ubuntu xenial-security/universe amd64 Packages [627 kB]
get:13 http://archive.ubuntu.com/ubuntu xenial/restricted Sources [5,179 B]
get:14 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9,802 kB]
```

图 10 Docker 镜像定制

```

Step 6/17 : ADD files/etc/stackstorm.com /etc/swift/etc/stackstorm.com
---> 6cc4554b51c9
Step 7/17 : ADD files/rsyncd.conf /etc/rsyncd.conf
---> a2ba6b9563a8
Step 8/17 : ADD files/swift.conf /etc/swift/swift.conf
---> d477e21338d6
Step 9/17 : ADD files/proxy-server.conf /etc/swift/proxy-server.conf
---> ba723c1d91ae
Step 10/17 : ADD files/account-server.conf /etc/swift/account-server.conf
---> c2b363a9c700
Step 11/17 : ADD files/object-server.conf /etc/swift/object-server.conf
---> 457a90643941
Step 12/17 : ADD files/container-server.conf /etc/swift/container-server.conf
---> 799bb6e75524
Step 13/17 : ADD files/proxy-server.conf /etc/swift/proxy-server.conf
---> 9a273a087c27
Step 14/17 : ADD files/startmain.sh /usr/local/bin/startmain.sh
---> 5a52716ea524
Step 15/17 : RUN chmod 755 /usr/local/bin/*.sh
---> Running in ef58cfb1c9ca
Removing intermediate container ef58cfb1c9ca
---> a0c49ebf99e3
Step 16/17 : EXPOSE 8080
---> Running in 7d3bacfd4124
Removing intermediate container 7d3bacfd4124
---> e48f534b2a52
Step 17/17 : CMD /usr/local/bin/startmain.sh
---> Running in b26315e46f24
Removing intermediate container b26315e46f24
---> c1e6c04602ec
Successfully built c1e6c04602ec
Successfully tagged openstack-swift-docker:latest
(base) wang@wanglinux:~/lab/openstack-swift-docker$

```

图 11 构建完毕

然后定制数据分卷：

```
sudo docker run -v /srv --name SWIFT_DATA busybox
```

此时本地环境已经准备完毕，首先启动一个容器实例，执行如下命令：

```
sudo docker run -d --name openstack-swift -p 12345:8080 --volumes-from SWIFT_DATA -t openstack-swift-docker
```

之后测试容器实例是否启动，执行如下命令，查看结果：

```
sudo docker logs openstack-swift
sudo docker ps
```

结果如下：

```

(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo docker logs openstack-swift
No existing ring files, creating them...
Device d0r1z1-127.0.0.1:6010R127.0.0.1:6010/sdb1_" with 1.0 weight got id 0
Reassigned 128 (100.00%) partitions. Balance is now 0.00. Dispersion is now 0.00
Device d0r1z1-127.0.0.1:6011R127.0.0.1:6011/sdb1_" with 1.0 weight got id 0
Reassigned 128 (100.00%) partitions. Balance is now 0.00. Dispersion is now 0.00
Device d0r1z1-127.0.0.1:6012R127.0.0.1:6012/sdb1_" with 1.0 weight got id 0
Reassigned 128 (100.00%) partitions. Balance is now 0.00. Dispersion is now 0.00
Copying ring files to /srv to save them if it's a docker volume...
Starting supervisord...

```

图 12 docker logs

```

(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS                NAMES
57ba549c7e2b   openstack-swift-docker   "/bin/sh -c /usr/loc..."   30 hours ago   Up 3 minutes   0.0.0.0:12345->8080/tcp   openstack-swift

```



图 13 查看 docker 是否启动

为了验证服务端功能的正确性，我们使用 python-swiftclient 来对服务端进行测试。  
执行：

```
sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing stat
```

有如下图的结果：

```
(base) wang@wanglinux:~$ sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing stat
Account: AUTH_test
Containers: 0
Objects: 0
Bytes: 0
Content-Type: text/plain; charset=utf-8
X-Timestamp: 1590632763.95819
X-Put-Timestamp: 1590632763.95819
X-Trans-Id: tx465a68f5d9a344b2906a5-005ecf213b
```

图 13 swift 客户端测试

如图所示，可以看到我们设置的用户名 test（在配置文件中），以及其他的属性信息，接着我们测试一下文件的存储和下载功能。

首先执行如下命令，尝试把本地的 1st 文件上传进入服务端中，并且命名为 test.txt;把本地的 test.ppt（其实就是实验 ppt，改成了这个名字，检测功能是否执行正确）上传，并命名为 iot-storage-experiment.pptx，并用 list 命令检查是否上传成功：

```
sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing
upload --object-name test.txt SWIFT_DATA ./1st
sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing
list SWIFT_DATA
sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing
upload --object-name iot-storage-experiment.pptx SWIFT_DATA ./test.pptx
sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing
list SWIFT_DATA
```

结果如下：

```
(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing upload --object-name test.txt SWIFT_DATA ./1st
test.txt
(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing list SWIFT_DATA
test.txt
(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing upload --object-name iot_storage_lab.pptx SWIFT_DATA ./test.pptx
iot_storage_lab.pptx
(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing list SWIFT_DATA
test.txt
iot_storage_lab.pptx
```

图 14 结果检验

接着尝试从对象存储服务端中下载这些文件，执行命令：

```
sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing
download SWIFT_DATA test.txt
sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing
download SWIFT_DATA iot-storage-experiment.pptx
```

如下图，成功验证了功能的正确性：

```
(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing download SWIFT_DATA iot_storage_lab.pptx
iot_storage_lab.pptx [auth 0.01s, headers 0.072s, total 0.396s, 22.530 MB/s]
(base) wang@wanglinux:~/lab/openstack-swift-docker$ sudo swift -A http://127.0.0.1:12345/auth/v1.0 -U test:tester -K testing download SWIFT_DATA test.txt
test.txt [auth 0.023s, headers 0.139s, total 0.140s, 0.000 MB/s]
(base) wang@wanglinux:~/lab/openstack-swift-docker$
```

图 15 下载文件

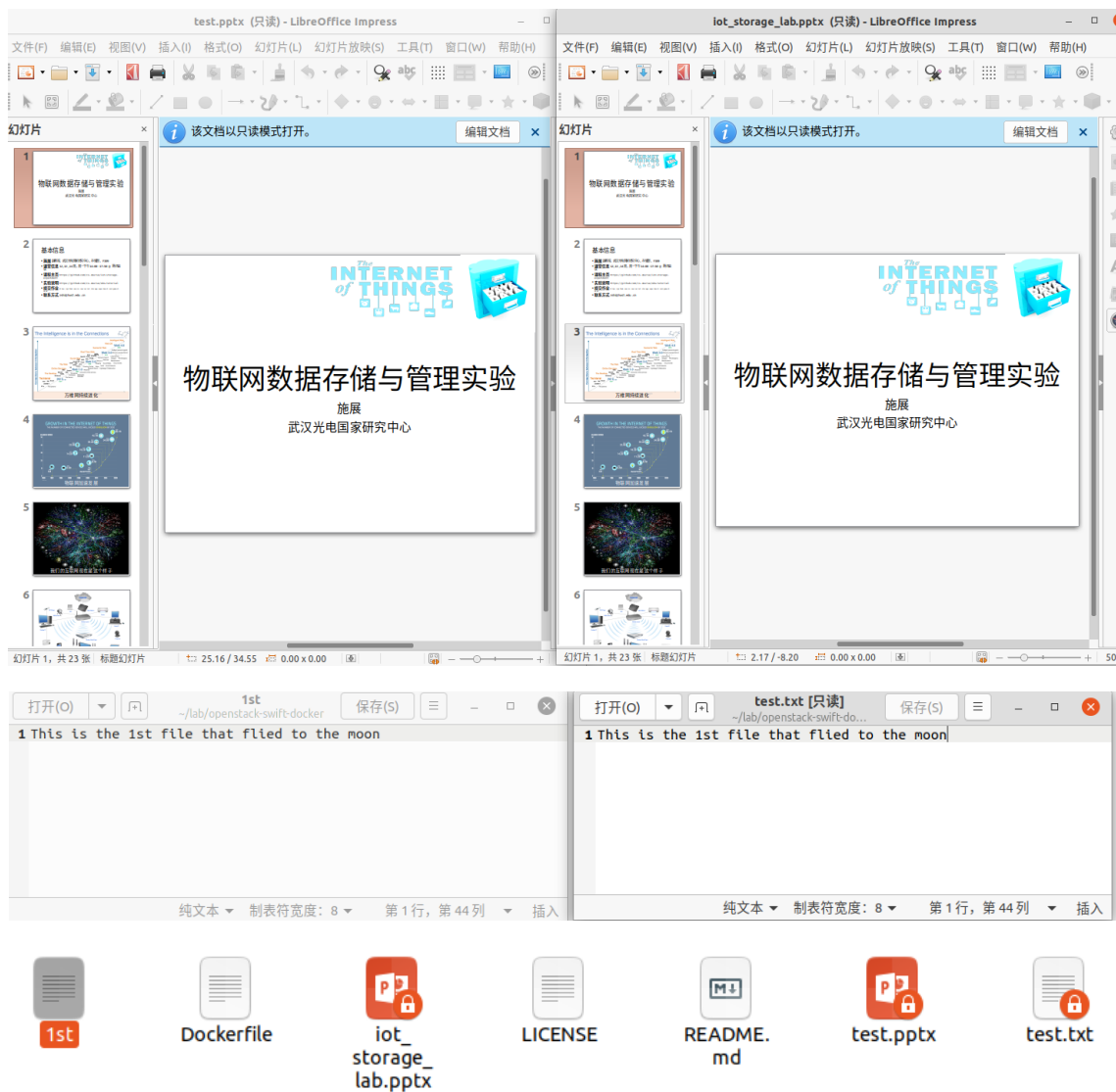


图 16, 17, 18 文件下载成功并检验一致性

### 5.3 对象存储性能分析

为了测试搭建的服务器的性能，以及服务器的百分位延迟，我们使用 `s3bench` 测试。但是由于它不像 `cosbench` 有批测试的功能。所以我们需要自己写脚本文件来测试。脚本进行批处理重定向到文件之后，再写一个小脚本将特定量的数字提取出来，然后导入到 `exel` 表格中，易于对数据分析处理。脚本代码如下：

```
endpoint="http://127.0.0.1:9000"
bucket="loadgen"
ObjectNamePrefix="loadgen"
AccessKey="hust"
AccessSecret="hust_obs"
filename="minio.txt

declare -a NumClient
declare -a NumSample
```

```

declare -a ObjectSize

NumClient=(1 1 1 1 1 1 1 1 1 1
1 1 2 4 8 16 32 64 70 80 90 100 2
4 8 16 32 64 1 1 1 1 1 1 1 1
1 )
NumSample=(100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 100 100 100 100 100 100 100 100
100 100 100 100 100 5 10 20 40 80 160 320 640
1280 )
ObjectSize=(1024 2048 4096 10240 20480 40960 102400 204800 409600 1048576
4194304 4096 4096 4096 4096 4096 4096 4096 4096 4096 4096 102400
102400 102400 102400 102400 102400 4096 4096 4096 4096 4096 4096 4096
4096 4096 )

for(( i=0;i<${#NumClient[@]};i++))
do
    # 命令
    ./s3bench -accessKey=$AccessKey -accessSecret=$AccessSecret -bucket=$
bucket -endpoint=$endpoint \ >> $filename
    -numClients=${NumClient[i]} -numSamples=${NumSample[i]} -objectNamePr
efix=$ObjectNamePrefix -objectSize=${ObjectSize[i]} | awk '($3=="99th" ||
$1=="Total" || $1=="Results" || $1=="Number"){ print}' >> $filename
    echo "-----"
    -----done">> $filename
done

```

对 minio 搭建的对象存储服务的测试结果如下:

### 1.对象尺寸对性能的影响

objectSize	w_throughput(mb/s)	w_duration(s)	w_99th(s)	r_throughput(mb/s)	r_duration(s)	r_99th(s)
1024	0.22	0.444	0.021	0.7	0.14	0.006
2048	0.27	0.737 s	0.131	1.95	0.1	0.003
4096	0.86	0.453	0.028	4.11	0.095	0.002
10240	2.07	0.472	0.036	10.67	0.092	0.002
20480	4.1	0.477	0.031	20.44	0.096	0.003
40960	7.13	0.548	0.084	37.97	0.103	0.002
102400	20.61	0.474	0.009	88.48	0.11	0.003
204800	35.17	0.555	0.022	156.4	0.125	0.003
409600	46.78	0.835	0.174	266.02	0.147	0.003
1048576	93.01	1.075	0.022	464.48	0.215	0.007
4194304	56.03	7.139	0.662	763.68	0.524	0.008

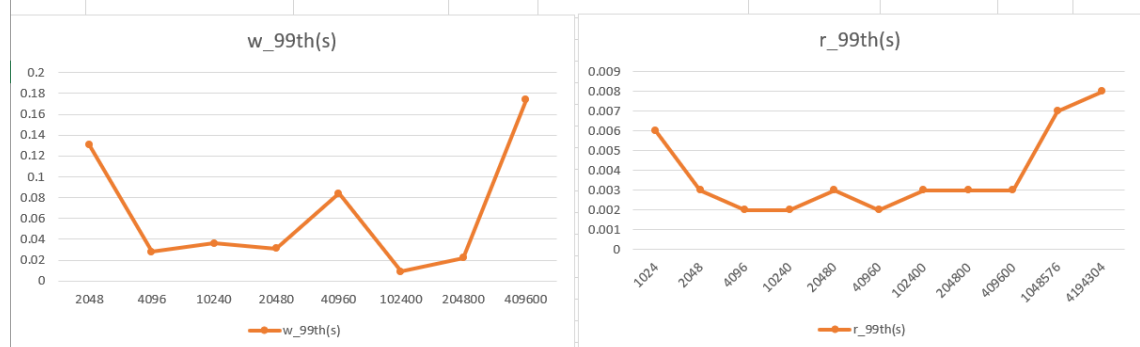


图 19 测试结果一

对象存储的大小从 1kb 增加到 4mb，并发客户端为 1，用于测试对象大小对对象存储系统的关系。可以看到，对象尺寸越大，吞吐率也越大。在 Minio 中，当对象尺寸为 4MB 时，写入速率可达到 56MB/s，读取速率可以达到 763MB/s。这个速率是服务器运行在本地的速率，对网络性能的影响可以忽略。实际应用中，吞吐率往往还与网络带宽速率有关。下面我们来看对象的百分位延迟。可以看到，无论是读或写，99%的对象延迟随着对象大小先减小后增大。

在任意一组数据中，读操作都要比写操作要快，本次实验将自己的个人 PC 作为服务器，PC 中磁盘由于自身结构导致读性能远远高于写性能。本来简单的以为对象尺寸越大，吞吐率越大，当增大到 4m 后读写的吞吐率开始下降，推测应该有个上限，而且之前已经达到或者接近了，阅读相关资料后分析为：当对象数据比较小时，磁盘读写的时间占比少，反而是读写的准备工作用时长，计算吞吐率时这些时间都要算进去；随着数据总量持续增长，磁盘读取和写入所花费的时间越来越多，吞吐率越来越接近磁盘的真实读取和写入速率。当读取和写入数据总量超过标准时，主存无法全部放下，需要将数据放入硬盘中，硬盘在读取和写入上远远慢于主存，将数据从硬盘调入主存中需要更多的时间，这就导致了吞吐率的下降。

## 2. 连接数量对性能的影响

将连接数量 numclients 设为 1 到 100 之间阶梯上涨：

objectSize	numClients	w_throughput(mb/s)	w_duration(s)	w_99th(s)	r_throughput(mb/s)	r_duration(s)	r_99th(s)
4096	1	0.27	1.459	0.48	3.57	0.109	0.005
4096	2	1.04	0.377	0.056	3.66	0.107	0.004
4096	4	1.63	0.24	0.046	3.95	0.099	0.076
4096	8	1.58	0.248	0.074	3.58	0.109	0.013
4096	16	1.99	0.197	0.098	3.84	0.102	0.101
4096	32	1.8	0.217	0.159	3.64	0.107	0.105
4096	64	1.26	0.309	0.283	3.05	0.128	0.121
4096	70	1.5	0.261	0.207	3.32	0.117	0.114
4096	80	0.13	2.946	2.943	3.16	0.124	0.114
4096	90	0.15	2.545	2.528	3.12	0.125	0.122
4096	100	0.22	1.816	1.812	2.88	0.136	0.116
102400	2	27.67	0.353	0.018	87.73	0.111	0.055
102400	4	34.88	0.28	0.04	88.35	0.111	0.008
102400	8	33.84	0.289	0.061	63.85	0.153	0.138
102400	16	37.31	0.262	0.144	82.89	0.118	0.041
102400	32	22.47	0.435	0.419	46.74	0.209	0.202
102400	64	32.77	0.298	0.286	73.4	0.133	0.122

图 20 测量结果结果 2

对两种对象大小，连接数越多，存储延迟越大，即使吞吐率上升，但是连接数多带来的每个存储对象的开销增多，使得服务器整体性能有一定下降。

## 3.对象数量 numsamples 的多少对性能的影响

objectSize	numSamples	w_throughput(mb/s)	w_duration(s)	w_99th(s)	r_throughput(mb/s)	r_duration(s)	r_99th(s)
4096	5	0.82	0.024	0.007	2.9	0.007	0.002
4096	10	0.97	0.04	0.005	2.85	0.014	0.005
4096	20	0.69	0.113	0.017	1.5	0.052	0.009
4096	40	0.37	0.42	0.139	3.53	0.044	0.002
4096	80	0.71	0.438	0.073	3.61	0.087	0.004
4096	160	0.58	1.071	0.114	3.72	0.168	0.003
4096	320	0.87	1.444	0.011	3.94	0.318	0.003
4096	640	0.61	4.082	0.046	3.73	0.67	0.003
4096	1280	0.76	6.567	0.032	3.94	1.269	0.002

图 21 测量结果 3

额，观察发现，numsamples 对性能的影响好像不明显，即使对象数量倍增，延迟也没怎么改变，猜测是数据传输给服务器是串行的原因，对服务器性能没什么影响。

## 六、实验总结

从上述测试结果和分析来看，在日常生活中，对读写延迟要求严格，需要它比较小的应用场景，比如查询和搜索，应该应用对象尺寸小的存储；而需要吞吐率与速度时，如网盘云盘或者内容提供商，应该选择对象尺寸大的对象存储。而延迟的影响因素里，对象大小的影响应该是来源于底层硬件磁盘读取的影响，连接数量是客户端并发请求，如果发生拥塞，需要排队处理连接请求的信息，增加了平均延迟。

首先一开始熟悉 `git` 和 `github` 的使用，由于之前用过（虽然之前主用 `gitee`），也对 `git` 有过了解，所以使用学习起来不难，而且这个对信息学科和写代码的同学来说掌握这个很重要。而配各种语言环境，由于已经在 `linux` 上做过很多实验，这个也不难，`anaconda` 之前做中间件的时候已经用到，而 `docker` 在中间件课程上有讲过，虽然这次实验简单地学习了部分使用（是真的强大而且方便），但是觉得 `docker` 学习起来还有很多。

本来搭建了 `swift` 想用 `cosbench` 来测试（数据整理的很好，自带批处理，很方便），结果显示因为没有仔细阅读老师的提示文档，`jdk` 版本和 `cosbench` 版本错误（群里也有同学提出来了，看了老师的提示后用 `docker` 来运行），后来修改配置文件后，发现读写操作的 `workers` 数量达到一定数量后总是中断且失败，如果用比较小的 `worker` 值又测不了几组，无奈只好放弃（不知道是不是因为服务端和测试客户端都在 `docker` 启动且本地主机的原因）。

实验过程中也体会到了从一个开源软件或者框架到实际应用中间差了很多东西，实际中各种服务器、各种集群应用或者不同的物理硬件环境，都会带来各种因素。

其实在以前学长的前人栽树下，以及老师上课详细完备地介绍（老师上课实验给的指导真的很详细很好），而且 `github` 上提供各种脚本，注释和资料链接，已经构成了大学以来最好最详细的实验指导和任务布置之一，感觉实验做的很舒服也有收获。配好了对象存储客户端之后，通过自己实际的操作，也明白了它的优势之处，了解到了以后可以用它来做点什么（比如个人简易网盘）。实验不长，收获了很多。

最后，非常感谢老师以及各种资料的提供者给予的帮助！

## 参考文献

- [1] ARNOLD J. OpenStack Swift[M]. O’ Reilly Media, 2014.
  - [2] ZHENG Q, CHEN H, WANG Y 等. COSBench: A Benchmark Tool for Cloud Object Storage Services[C]//2012 IEEE Fifth International Conference on Cloud Computing. 2012: 998 - 999.
  - [3] WEIL S A, BRANDT S A, MILLER E L 等. Ceph: A Scalable, High-performance Distributed File System[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Berkeley, CA, USA: USENIX Association, 2006: 307 - 320.
- （可以根据实际需要更新调整）